Hadoop Streaming 编程

源地址：

http://dongxicheng.org/mapreduce/hadoop-streaming-programming/

1、概述

Hadoop Streaming 是 Hadoop 提供的一个编程工具，它允许用户使用任何可执行文件或者脚本文件作为 Mapper 和 Reducer，例如：

采用 shell 脚本语言中的一些命令作为 mapper 和 reducer （cat 作为 mapper，wc 作为 reducer）

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/hadoop-streaming.jar \
-input myInputDirs \
-output myOutputDir \
-mapper cat \
-reducer wc
```

本文安排如下，第二节介绍 Hadoop Streaming 的原理，第三节介绍 Hadoop Streaming 的使用方法，第四节介绍 Hadoop Streaming 的程序编写方法，在这一节中，用 C++、C、shell 脚本和 python 实现了 WordCount 作业，第五节总结了常见的问题。文章最后给出了程序下载地址。(本文内容基于 Hadoop-0.20.2 版本)

(注：如果你采用的语言为 C 或者 C++，也可以使用 Hadoop Pipes，具体可参考这篇文章：Hadoop Pipes 编程。)

关于 Hadoop Streaming 高级编程方法，可参考这篇文章：Hadoop Streaming 高级编程。

2、Hadoop Streaming 原理

mapper 和 reducer 会从标准输入中读取用户数据，一行一行处理后发送给标准输出。Streaming 工具会创建 MapReduce 作业，发送给各个 tasktracker，同时监控整个作业的执行过程。

如果一个文件（可执行或者脚本）作为 mapper，mapper 初始化时，每一个 mapper 任务会把该文件作为一个单独进程启动，mapper 任务运行时，它把输入切分成行并把每一行提供给可执行文件进程的标准输入。同时，mapper 收集可执行文件进程标准输出的内容，并把收到的每一行内容转化成 key/value 对，作为 mapper 的输出。默认情况下，一行中第一个 tab 之前的部分作为 key，之后的（不包括 tab）作为 value。如果没有 tab，整行作为 key 值，value 值为 null。

对于 reducer，类似。

以上是 Map/Reduce 框架和 streaming mapper/reducer 之间的基本通信协议。

3、Hadoop Streaming 用法

Usage: $HADOOP_HOME/bin/hadoop jar \
$HADOOP_HOME/hadoop-streaming.jar [options]

options：

（1）-input：输入文件路径

（2）-output：输出文件路径

（3）-mapper：用户自己写的 mapper 程序，可以是可执行文件或者脚本

（4）-reducer：用户自己写的 reducer 程序，可以是可执行文件或者脚本

（5） -file ：打包文件到提交的作业中，可以是 mapper 或者 reducer 要用的输入文件，如配置文件，字典等。

（6） -partitioner ：用户自定义的 partitioner 程序

（7） -combiner ：用户自定义的 combiner 程序（必须用 java 实现）

（8） -D ：作业的一些属性（以前用的是 -jonconf ），具体有：

1） mapred.map.tasks ：map task 数目

2） mapred.reduce.tasks ：reduce task 数目

3） stream.map.input.field.separator/stream.map.output.field.separator ：map task 输入/输出数据的分隔符，默认均为 \t 。

4） stream.num.map.output.key.fields ：指定 map task 输出记录中 key 所占的域数目

5） stream.reduce.input.field.separator/stream.reduce.output.field.separator ：reduce task 输入/输出数据的分隔符，默认均为 \t 。

6） stream.num.reduce.output.key.fields ：指定 reduce task 输出记录中 key 所占的域数目

另外，Hadoop本身还自带一些好用的 Mapper和 Reducer：

（1） Hadoop聚集功能

Aggregate 提供一个特殊的 reducer 类和一个特殊的 combiner 类，并且有一系列的“聚合器”（例如“sum”，“max”，“min”等）用于聚合一组 value 的序列。用户可以使用 Aggregate 定义一个 mapper 插件类，这个类用于为 mapper 输入的每个 key/value 对产生“可聚合项”。Combiner/reducer 利用适当的聚合器聚合这些可聚合项。要使用 Aggregate ，只需指定“-reducer aggregate ”。

（2）字段的选取（类似于 Unix 中的‘cut’）

Hadoop 的工具类 org.apache.hadoop.mapred.lib.FieldSelectionMapReduc 帮助用户高效处理文本数据，就像 unix 中的“cut”工具。工具类中的 map 函数把输入的 key/value 对看作字段的列表。用户可以指定字段的分隔符（默认是 tab ），可以选择字段列表中任意一段（由列表中一个或多个字段组成）作为 map 输出的 key 或者 value 。同样，工具类中的 reduce 函数也把输入的 key/value 对看作字段的列表，用户可以选取任意一段作为 reduce 输出的 key 或 value 。

4、Mapper 和 Reducer 实现

本节试图用尽可能多的语言编写 Mapper 和 Reducer ，包括 Java，C，C++ ，Shell 脚本，python 等。

由于 Hadoop 会自动解析数据文件到 Mapper 或者 Reducer 的标准输入中，以供它们读取使用，所有应先了解各个语言获取标准输入的方法。

程序见 Hadoop streaming.rar

5、常见问题

（1）作业总是运行失败：

需要把 mapper 文件和 reducer 文件放到各个 tasktracker 上，保证各个节点均有一份。也可在提交作业时，采用 -file 选项指定这些文件。

（2）用脚本编写时，第一行需注明脚本解释器，默认是 shell

6、参考资料

【1】C++&Python 实现 Hadoop Streaming 的 paritioner 和模块化

【2】如何在 Hadoop 中使用 Streaming 编写 MapReduce

【3】Hadoop 如何与 C++ 结合

【4】Hadoop Streaming 和 pipes 理解

下面是一个用 C++ 写的 WordCount 示例：

http://bbs.hadoopor.com/thread-420-1-2.html

```
#include "hadoop/Pipes.hh"
#include "hadoop/TemplateFactory.hh"
#include "hadoop/StringUtils.hh"

class WordCountMap: public HadoopPipes::Mapper {
public:
  WordCountMap(HadoopPipes::TaskContext& context){}
  void map(HadoopPipes::MapContext& context) {
    std::vector<std::string> words =
      HadoopUtils::splitString(context.getInputValue(), " ");
    for(unsigned int i=0; i < words.size(); ++i) {
      context.emit(words[i], "1");
    }
  }
};

class WordCountReduce: public HadoopPipes::Reducer {
public:
  WordCountReduce(HadoopPipes::TaskContext& context){}
  void reduce(HadoopPipes::ReduceContext& context) {
```

```
    int sum = 0;
    while (context.nextValue()) {
      sum += HadoopUtils::toInt(context.getInputValue());
    }
    context.emit(context.getInputKey(), HadoopUtils::toString(sum));
  }
};

int main(int argc, char *argv[]) {
  return HadoopPipes::runTask(HadoopPipes::TemplateFactory<WordCountMap,
                              WordCountReduce>());
}
```
复制代码

编译，生成可执行文件 ：

```
# ant -Dcompile.c++=yes examples
```

将可执行文件编译上传到 HDFS ：

bin/hadoop fs -put build/c++-examples/Linux-i386-32/bin /examples/bin

创建配置文件：

```
# vi src/examples/pipes/conf/word.xml

<?xml version="1.0"?>
<configuration>
  <property>
    // Set the binary path on DFS
    <name>hadoop.pipes.executable</name>
    <value>/examples/bin/wordcount</value>
  </property>
  <property>
    <name>hadoop.pipes.java.recordreader</name>
    <value>true</value>
```

```
    </property>
    <property>
      <name>hadoop.pipes.java.recordwriter</name>
      <value>true</value>
    </property>
  </configuration>
```

复制代码

运行示例：
bin/hadoop pipes -conf src/examples/pipes/conf/word.xml -input in-dir -output out-dir

如何在 Hadoop 中使用 Streaming 编写 MapReduce
http://bbs.hadoopor.com/thread-256-1-1.html
作者：马士华 发表于： 2008-03-05 12:51 最后更新于： 2008-03-25 11:18
版权声明：可以任意转载，转载时请务必以超链接形式标明文章原始出处和作者信息。
http://www.hadoop.org.cn/hadoop/hadoop-streaming/

Michael G. Noll 在他的 Blog 中提到如何在 Hadoop 中用 Python 编写 MapReduce 程序，韩国的 gogamza 在其 Bolg 中也提到如何用 C 编写 MapReduce 程序（我稍微修改了一下原程序，因为他的 Map 对单词切分使用 tab 键）。我合并他们两人的文章，也让国内的 Hadoop 用户能够使用别的语言来编写 MapReduce 程序。
首先您得配好您的 Hadoop 集群，这方面的介绍网上比较多，这儿给个链接（Hadoop 学习笔记二 安装部署）。Hadoop Streaming 帮助我们用非 Java 的编程语言使用 MapReduce，Streaming 用 STDIN (标准输入)和 STDOUT (标准输出)来和我们编写的 Map 和 Reduce进行数据的交换数据。任何能够使用 STDIN和 STDOUT都可以用来编写 MapReduce 程序，比如我们用 Python 的 sys.stdin 和 sys.stdout，或者是 C 中的 stdin 和 stdout。
我们还是使用 Hadoop 的例子 WordCount 来做示范如何编写 MapReduce，在 WordCount 的例子中我们要解决计算在一批文档中每一个单词的出现频率。首先我们在 Map 程序中会接受到这批文档每一行的数据，然后我们编写的 Map 程序把这一行按空格切开成一个数组。并对这个数组遍历按 "<word> 1" 用标准的输出输出来，代表这个单词出现了一次。在 Reduce 中我们来统计单词的出现频率。

Python Code
Map: mapper.py
#!/usr/bin/env python

```
import sys

# maps words to their counts
word2count = {}

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words while removing any empty strings
    words = filter(lambda word: word, line.split())
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

复制代码

Reduce: reducer.py

```
#!/usr/bin/env python

from operator import itemgetter
import sys

# maps words to their counts
word2count = {}

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word, count = line.split()
    # convert count (currently a string) to int
    try:
        count = int(count)
        word2count[word] = word2count.get(word, 0) + count
```

```
        except ValueError:
                # count was not a number, so silently
                # ignore/discard this line
                pass

# sort the words lexigraphically;
#
# this step is NOT required, we just do it so that our
# final output will look more like the official Hadoop
# word count examples
sorted_word2count = sorted(word2count.items(), key=itemgetter(0))

# write the results to STDOUT (standard output)
for word, count in sorted_word2count:
        print '%s\t%s'% (word, count)
```

复制代码

C CodeMap: Mapper.c

```
#include <stdio.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define BUF_SIZE            2048
#define DELIM       "\n"

int main(int argc, char *argv[]){
        char buffer[BUF_SIZE];
        while(fgets(buffer, BUF_SIZE - 1, stdin)){
                int len = strlen(buffer);
                if(buffer[len-1] == '\n')
                  buffer[len-1] = 0;

                char *querys     = index(buffer, ' ');
                char *query = NULL;
                if(querys == NULL) continue;
                querys += 1; /*     not to include '\t' */

                query = strtok(buffer, " ");
                while(query){
                        printf("%s\t1\n", query);
                        query = strtok(NULL, " ");
                }
```

```
        }
        return 0;
}
```

复制代码

Reduce: Reducer.c

```
#include <stdio.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFFER_SIZE       1024
#define DELIM      "\t"

int main(int argc, char *argv[]){
    char strLastKey[BUFFER_SIZE];
    char strLine[BUFFER_SIZE];
    int count = 0;

    *strLastKey = '\0';
    *strLine = '\0';

    while( fgets(strLine, BUFFER_SIZE - 1, stdin) ){
        char *strCurrKey = NULL;
        char *strCurrNum = NULL;

        strCurrKey    = strtok(strLine, DELIM);
        strCurrNum = strtok(NULL, DELIM); /* necessary to check error but.... */

        if( strLastKey[0] == '\0'){
            strcpy(strLastKey, strCurrKey);
        }

        if(strcmp(strCurrKey, strLastKey)){
            printf("%s\t%d\n", strLastKey, count);
            count = atoi(strCurrNum);
        }else{
            count += atoi(strCurrNum);
        }
        strcpy(strLastKey, strCurrKey);

    }
    printf("%s\t%d\n", strLastKey, count); /* flush the count */
```

```
    return 0;
}
```
复制代码

首先我们调试一下源码：

```
chmod +x mapper.py
chmod +x reducer.py
echo "foo foo quux labs foo bar quux" | ./mapper.py | ./reducer.py
bar	1
foo	3
labs	1
quux	2
g++ Mapper.c -o Mapper
g++ Reducer.c -o Reducer
chmod +x Mapper
chmod +x Reducer
echo "foo foo quux labs foo bar quux" | ./Mapper | ./Reducer
bar	1
foo	2
labs	1
quux	1
foo	1
quux	1
```
复制代码

你可能看到 C 的输出和 Python 的不一样 ,因为 Python 是把他放在词典里了 .我们在 Hadoop 时,会对这进行排序 ,然后相同的单词会连续在标准输出中输出 .在 Hadoop 中运行程序首先我们要下载我们的测试文档 wget http://www.gutenberg.org/dirs/etext04 /7ldvc10.txt. 我们把文档存放在 /tmp/doc 这个目录下 .拷贝测试文档到 HDFS中.

```
bin/hadoop dfs -copyFromLocal /tmp/doc doc
```
复制代码

运行 bin/hadoop dfs -ls doc 看看拷贝是否成功 .接下来我们运行我们的 MapReduce 的 Job.

```
bin/hadoop jar /home/hadoop/contrib/hadoop-0.15.1-streaming.jar -mapper /home/hadoop/Mapper\
-reducer /home/hadoop/Reducer -input doc/* -output c-output -jobconf mapred.reduce.tasks=1

additionalConfSpec_:null
null=@@@userJobConfProps_.get(stream.shipped.hadoopstreaming
packageJobJar: [] [/home/msh/hadoop-0.15.1 /contrib/hadoop-0.15.1-streaming.jar] /tmp/streamjob60816.jar tmpDir=null
```

08/03/04 19:03:13 INFO mapred.FileInputFormat: Total input paths to process : 1
08/03/04 19:03:13 INFO streaming.StreamJob: getLocalDirs(): [/home/msh/data/filesystem/mapred/local]
08/03/04 19:03:13 INFO streaming.StreamJob: Running job: job_200803031752_0003
08/03/04 19:03:13 INFO streaming.StreamJob: To kill this job, run:
08/03/04 19:03:13 INFO streaming.StreamJob: /home/msh/hadoop/bin/../bin/hadoop job -Dmapred.job.tracker=192.168.2.92:9001 -kill job_200803031752_0003
08/03/04 19:03:13 INFO streaming.StreamJob: Tracking URL: http://hadoop-master:50030/jobdetails.jsp?jobid=job_200803031752_0003
08/03/04 19:03:14 INFO streaming.StreamJob: map 0% reduce 0%
08/03/04 19:03:15 INFO streaming.StreamJob: map 33% reduce 0%
08/03/04 19:03:16 INFO streaming.StreamJob: map 100% reduce 0%
08/03/04 19:03:19 INFO streaming.StreamJob: map 100% reduce 100%
08/03/04 19:03:19 INFO streaming.StreamJob: Job complete: job_200803031752_0003
08/03/04 19:03:19 INFO streaming.StreamJob: Output: c-output

bin/hadoop jar /home/hadoop/contrib/hadoop-0.15.1-streaming.jar -mapper /home/hadoop/mapper.py\
-reducer /home/hadoop/reducer.py -input doc/* -output python-output -jobconf mapred.reduce.tasks=1

additionalConfSpec_:null
null=@@@userJobConfProps_.get(stream.shipped.hadoopstreaming
packageJobJar: [] [/home/hadoop/hadoop-0.15.1 /contrib/hadoop-0.15.1-streaming.jar] /tmp/streamjob26099.jar tmpDir=null
08/03/04 19:05:40 INFO mapred.FileInputFormat: Total input paths to process : 1
08/03/04 19:05:41 INFO streaming.StreamJob: getLocalDirs(): [/home/msh/data/filesystem/mapred/local]
08/03/04 19:05:41 INFO streaming.StreamJob: Running job: job_200803031752_0004
08/03/04 19:05:41 INFO streaming.StreamJob: To kill this job, run:
08/03/04 19:05:41 INFO streaming.StreamJob: /home/msh/hadoop/bin/../bin/hadoop job -Dmapred.job.tracker=192.168.2.92:9001 -kill job_200803031752_0004
08/03/04 19:05:41 INFO streaming.StreamJob: Tracking URL: http://hadoop-master:50030/jobdetails.jsp?jobid=job_200803031752_0004
08/03/04 19:05:42 INFO streaming.StreamJob: map 0% reduce 0%
08/03/04 19:05:48 INFO streaming.StreamJob: map 33% reduce 0%
08/03/04 19:05:49 INFO streaming.StreamJob: map 100% reduce 0%
08/03/04 19:05:52 INFO streaming.StreamJob: map 100% reduce 100%
08/03/04 19:05:52 INFO streaming.StreamJob: Job complete: job_200803031752_0004
08/03/04 19:05:52 INFO streaming.StreamJob: Output: python-output
复制代码

当 Job 提交后我们还能够在 web 的界面 http://localhost:50030/ 看到每一个工作的运行情况。
当 Job 工作完成后我们能够在 c-output 和 python-output 看到一些文件

bin/hadoop dfs -ls c-output
复制代码

输入下面的命令我们能够看到我们运行完 MapReduce 的结果

bin/hadoop dfs -cat c-output/part-00000
复制代码

用 Hadoop Streaming 运行 MapReduce 会比较用 Java的代码要慢 ,因为有两方面的原因 :
使用 Java API >> C Streaming >> Perl Streaming这样的一个流程运行会阻塞 IO.
不像 Java在运行 Map 后输出结果有一定数量的结果集就启动 Reduce的程序 ,用 Streaming 要等到所有的 Map 都运行完毕后才启动 Reduce
如果用 Python 编写 MapReduce 的话，另一个可选的是使用 Jython 来转编译 Pyhton 为 Java 的原生码. 另外对于 C程序员更好的选择是使用 Hadoop 新的 C++ MapReduce API Pipes来编写. 不管怎样，毕竟 Hadoop 提供了一种不使用 Java 来进行分布式运算的方法. 下面是从 http://www.lunchpauze.com/ 2007/10/writing-hadoop-mapreduce-program-in-php.html 页面中摘下的用 php 编写的 MapReduce 程序 ,供 php 程序员参考：

Map: mapper.php

```
#!/usr/bin/php
<?

$word2count = array();

// input comes from STDIN (standard input)
while (($line = fgets(STDIN)) !== false) {
	// remove leading and trailing whitespace and lowercase
	$line = strtolower(trim($line));
	// split the line into words while removing any empty string
	$words = preg_split('/\W/', $line, 0, PREG_SPLIT_NO_EMPTY);
	// increase counters
	foreach ($words as $word) {
		$word2count[$word] += 1;
	}
}

// write the results to STDOUT (standard output)
// what we output here will be the input for the
// Reduce step, i.e. the input for reducer.py
foreach ($word2count as $word => $count) {
	// tab-delimited
	echo $word, chr(9), $count, PHP_EOL;
}

?>
```

复制代码

Reduce: mapper.php

```
#!/usr/bin/php
<?

$word2count = array();

// input comes from STDIN
while (($line = fgets(STDIN)) !== false) {
	// remove leading and trailing whitespace
	$line = trim($line);
	// parse the input we got from mapper.php
	list($word, $count) = explode(chr(9), $line);
	// convert count (currently a string) to int
	$count = intval($count);
	// sum counts
	if ($count > 0) $word2count[$word] += $count;
}

// sort the words lexigraphically
//
// this set is NOT required, we just do it so that our
// final output will look more like the official Hadoop
// word count examples
ksort($word2count);

// write the results to STDOUT (standard output)
foreach ($word2count as $word => $count) {
	echo $word, chr(9), $count, PHP_EOL;
}

?>
```

复制代码

c++ && python 实现 Hadoop Streaming 的 partitioner 和 模块化

http://www.cppblog.com/MemoryGarden/archive/ 2010/01/24/106312.html

这些东西是我自己的理解, 如果有错误的地方, 或者有哪些地方走了弯路, 请帮我指出我的错误, 谢谢

Hadoop Streaming 是一个工具, 代替编写 Java 的实现类, 而利用可执行程序来完成

map-reduce 过程

工作流程 ：

InputFile --> mappers --> [Partitioner] --> reducers --> outputFiles

理解 ：

1 输入文件，可以是指定远程文件系统内的文件夹下的 *

2 通过集群自己分解到各个 PC 上，每个 mapper 是一个可执行文件，相应的启动一个进程，来实现你的逻辑

3 mapper 的输入为标准输入，所以，任何能够支持标准输入的可执行的东西， c,c++(编译出来的可执行文件),python,..... 都可以作 为 mapper 和 reducer mapper 的输出为标准输出， 如果有 Partitioner，就给它，如果没有，它的输出将作为 reducer 的输入

4 Partitioner 为可选的项，二次排序，可以对结果进行分类打到结果文件里面 ,它的输入是 mapper 的标准输出，它的输出，将作为 reducer 的标准输入

5 reducer 同 mapper

6 输出文件夹，在远端文件不能重名

Hadoop Streaming

1 ： hadoop-streaming.jar 的位置 ： $HADOOP_HOME/contrib/streaming 内

官方上面关于 hadoop-streaming 的介绍已经很详细了，而且也有了关于 python 的例子，我就不说了 ,这里总结下自己的经验

1 指定 mapper or reducer 的 task 官方上说要用 -jobconf 但是这个参数已经过时，不可以用了， 官方说要用 -D，注意这个 -D 是要作为最开始的配置出现的， 因为是在 maper 和 reducer 执行之前，就需要硬性指定好的，所以要出现在参数的最前面 ./bin/hadoop jar hadoop-0.19.2-streaming.jar -D .....-input ..... 类似这样，这样，即使你程序最后只指定了一个输出管道，但是还是会有你指定的 task 数量的结果文件，只不过多余的就是空的 实验以下 就知道了

2 关于二次排序，由于是用的 streaming 所以，在可执行文件内，只能够处理逻辑，还有就是输出，当然我们也可以指定二次排序，但是由于是全部参数化，不是很灵活。比如 :

10.2.3.40 1

11.22.33.33 1

www.renren.com 1

www.baidu.com 1

10.2.3.40 1

这样一个很规整的输入文件，需求是要把记录独立的 ip 和 url 的 count 但是输出文件要分分割出来。

官方网站的例子，是指定 key 然后对 key 指定 主-key 和 key 用来排序，而 主-key 用来

二次排序，这样会输出你想要的东西，但是对于上面最简单的需求，对于传递参数，我们如何做呢？

其实我们还是可以利用这一点，在我们 mapper 里面，还是按照 /t 来分割 key value 但是我们要给 key 指定一个主 -key 用来给 Partitioner 来实现二次排序，所以我们可以稍微处理下这个 KEY,我们可以简单的判断出来 ip 和 url 的区别，这样，我们就人为的加上一个主 -key 我们在 mapper 里面，给每个 key 人为的加上一个 "标签 "，用来给 partitioner 做 二次排序用，比如我们的 mapper 的输出是这样

D&10.2.3.40 1
D&11.22.33.33 1
W&www.renren.com 1
W&www.baidu.com 1
D&10.2.3.40 1

然后通过传递命令参数

-partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner // 指定要求二次排序
-jobconf map.output.key.field.separator='&' // 这里如果不加两个单引号的话我的命令会死掉
-jobconf num.key.fields.for.partition=1 // 这里指第一个 & 符号来分割，保证不会出错

这样我们就可以通过 partitioner 来实现二次排序了

在 reducer 里面，我们再把 "标签" 摘掉 (不费吹灰之力) 就可以做到悄无声息的完成二次排序了。

3: 关于模块化

(强调 ： 没有在集群上测试，只在单机上做测试)

程序员最悲剧的就是不能代码复用，做这个也一样，用 hadoop-streaming 也一样，要做到代码重用，是我第一个考虑的问题
当我看到 -file(详细可以看官方网站上的讲解) 的时候，我就想到利用这个东西，果然，我的在本机上建立了一个 py 模块，简单的一个函数
然后在我的 mapper 里面 import 它，本地测试通过后，利用 -file 把模块所在的问价夹用 -file moudle/* 这个参数，传入 streaming
执行的结果毫无错误， 这样， 我们就可以抽象出来一些模块的东西， 来实现我们模块化的需求

注： 不要忘记 chmod +x *.py 将 py 变成可执行的，不然不可以运行

代码：

1: 模块代码 mg.py 用来给 mapper 贴标签

```
def mgFunction(line):
    if(line[0] >= '0' and line[0] <= '9'):
        return "D&" + line
    return "W&" + line
```

2: mapper.py

```
#!/usr/bin/env python
import sys
sys.path.append('/home/liuguoqing/Desktop/hadoop-0.19.2/moudle')
import mg
for line in sys.stdin:
    line = mg.mgFunction(line)
    line = line.strip()
#   print line
    words = line.split()
    print '%s\t%s' % (words[0], words[1])
```

3: reducer.py

```
#!/usr/bin/env python
import sys
user_login_day = {}

for line in sys.stdin:
    line = line[2:]//    去掉帽子
    line = line.strip()
    userid, day = line.split('\t', 1)
    user_login_day[userid] = user_login_day.get(userid, 0) + 1

for uid in user_login_day.keys():
    print '%s\t%d' % (uid, user_login_day[uid])
```

这样就实现了模块化的可以二次排序的 hadoop-streaming

命令

```
./bin/hadoop jar hadoop-0.19.2-streaming.jar \
#streaming jar
-D mapred.reduce.tasks=2          \
# 指定  2 个  reduce  来处理
-input user_login_day-input2/*            \
# 指定输入文件        可以用    dir/*      方式
-output user_login_day-output102
# 指定输出文件夹
-mapper ~/Desktop/hadoop-0.19.2/python/mapper/get_user_login_day_back.py                    \
# 指定  mapper      可执行文件     我用全路径，好像用相对路径会出错         ...
-reducer ~/Desktop/hadoop-0.19.2/python/reducer/get_user_login_day_back.py \
# 指定  reducer    可执行文件
-file ~/Desktop/hadoop-0.19.2/moudle/* \
# 指定模块化的库文件        dir/*       模式
-partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner \
# 指定    partitioner       参数为   class
-jobconf map.output.key.field.separator='&' \
# 指定      主-key      的分割符号为      '&'
-jobconf num.key.fields.for.partition=1
# 指定为第一个    ‘  & ’
```

```
liuguoqing@liuguoqing-desktop:~/Desktop/hadoop-0.19.2$ ./bin/hadoop jar
hadoop-0.19.2-streaming.jar -D mapred.reduce.tasks=2 -input user_login_day-input2/*
-output user_login_day-output102 -mapper
~/Desktop/hadoop-0.19.2/python/mapper/get_user_login_day_back.py -reducer
~/Desktop/hadoop-0.19.2/python/reducer/get_user_login_day_back.py -file
~/Desktop/hadoop-0.19.2/moudle/* -partitioner
org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner -jobconf
map.output.key.field.separator='&' -jobconf num.key.fields.for.partition=1
10/01/24 03:19:15 WARN streaming.StreamJob: -jobconf option is deprecated, please use -D
instead.
packageJobJar: [/home/liuguoqing/Desktop/hadoop-0.19.2/moudle/mg.py,
/home/liuguoqing/Desktop/hadoop-0.19.2/moudle/mg.pyc,
/tmp/hadoop-liuguoqing/hadoop-unjar6780057097425964518/] []
/tmp/streamjob3100401358387519950.jar tmpDir=null
10/01/24 03:19:15 INFO mapred.FileInputFormat: Total input paths to process : 2
10/01/24 03:19:15 INFO streaming.StreamJob: getLocalDirs():
[/tmp/hadoop-liuguoqing/mapred/local]
10/01/24 03:19:15 INFO streaming.StreamJob: Running job: job_201001221008_0065
10/01/24 03:19:15 INFO streaming.StreamJob: To kill this job, run:
10/01/24 03:19:15 INFO streaming.StreamJob:
/home/liuguoqing/Desktop/hadoop-0.19.2/bin/../bin/hadoop
job    -Dmapred.job.tracker=hdfs://localhost:9881 -kill job_201001221008_0065
```

```
10/01/24 03:19:15 INFO streaming.StreamJob: Tracking URL:
http://localhost:50030/jobdetails.jsp?jobid=job_201001221008_0065
10/01/24 03:19:16 INFO streaming.StreamJob:          map 0%     reduce 0%
10/01/24 03:19:17 INFO streaming.StreamJob:          map 33%     reduce 0%
10/01/24 03:19:18 INFO streaming.StreamJob:          map 67%     reduce 0%
10/01/24 03:19:19 INFO streaming.StreamJob:          map 100%     reduce 0%
10/01/24 03:19:27 INFO streaming.StreamJob:          map 100%     reduce 50%
10/01/24 03:19:32 INFO streaming.StreamJob:          map 100%     reduce 100%
10/01/24 03:19:32 INFO streaming.StreamJob: Job complete: job_201001221008_0065
10/01/24 03:19:32 INFO streaming.StreamJob: Output: user_login_day-output102
liuguoqing@liuguoqing-desktop:~/Desktop/hadoop-0.19.2$ ./bin/hadoop dfs -ls
user_login_day-output102
Found 3 items
drwxr-xr-x       - liuguoqing supergroup                  0 2010-01-24 03:19
/user/liuguoqing/user_login_day-output102/_logs
-rw-r--r--       1 liuguoqing supergroup                 25 2010-01-24 03:19
/user/liuguoqing/user_login_day-output102/part-00000
-rw-r--r--       1 liuguoqing supergroup                 47 2010-01-24 03:19
/user/liuguoqing/user_login_day-output102/part-00001

liuguoqing@liuguoqing-desktop:~/Desktop/hadoop-0.19.2$ ./bin/hadoop dfs -cat
user_login_day-output102/part-00000
54321      2
99999      1
12345      12
liuguoqing@liuguoqing-desktop:~/Desktop/hadoop-0.19.2$ ./bin/hadoop dfs -cat
user_login_day-output102/part-00001
http://www.renren.com           3
http://www.baidu.com           3
```

以上为操作结果显示

4 : c++ 的应用

只要写两个个标准输入输出的 mapper reducer ，然后

```
g++ mapper.cpp -o mapper
g++ reducer.cpp -o reducer
```

生成的两个可执行的 mapper reducer 的文件作为 mapper 和 reducer 参数就可以了， 执行的命令和上面是一样的

代码 ：

mapper.cpp

```
#include <stdio.h>
#include <string>
#include <iostream>
using namespace std;

int main(){
	string key;
	string value;
	while(cin>>key){
		cin>>value;
		cout<<key<<"\t"<<value<<endl;
	}
	return 0;
}
```

reducer.cpp

```
#include <stdio.h>
#include <string>
#include <map>
#include <iostream>
using namespace std;
int main(){
	string key;
	string value;
	map<string, int> word2count;
	map<string, int> :: iterator it;
	while(cin>>key){
		cin>>value;
		it = word2count.find(key);
		if(it != word2count.end()){
			++it->second;
		}
		else{
			word2count.insert(make_pair(key, 1));
			it->second = 0;
		}
	}

	for(it = word2count.begin(); it != word2count.end(); ++it){
		cout<<it->first<<"\t"<<it->second<<endl;
	}
```

```
    return 0;
}
```

这样就可以利用 c++ 来编写 hadoop map-reduce 了。

Hadoop Streaming 和 Pipes 理解

原始问题的引出见： http://bbs.hadoopor.com/viewthread.php?tid=542

在论坛里搜索了下 ,发现了两篇利用 C/C++ 编写 mapreduce 的文章 .如下 :

http://bbs.hadoopor.com/thread-256-1-1.html

http://bbs.hadoopor.com/thread-420-1-2.html

一. 其中 ,利用 stream 的文章中提到 ,利用 stream 编写 mapreduce 程序 ,其 reduce 任务要等到所有 map 任务结束后才能进行 ,这点不太理解 .

二. 从两种方法的实现上来看 .感觉有点奇怪 .在 linux 下 ,一般从 stdin 读入数据认为是管道的方式 ,而通过 socket 读入数据为 stream 方式 ,但是在 hadoop 中 ,貌似其叫法与 linux 下相反 . 不知道为什么 .

三. 从代码可以看出 ,在 hadoop 中 ,stream 是利用 stdin, 而 pipes 是利用 socket. 这两者有什么优缺点可以比较下的 .

By ：guxiangxi

一，二两个问题我也不理解，不止如何回答。 第三个问题我更关注一些，因为之前用过一下 streaming ，感觉不是特别好用，现在的状况还是熟悉 C++ ，但写 mapreduce 还是用 java 。正好 Pipes 正好是我想要的。下面有三处文章可以参考

1、 http://cxwangyi.blogspot.com/2010/01/writing-hadoop-programs-using-c.html

2、 http://remonstrate.wordpress.com/2010/10/01/hadoop-%E4%B8%8A%E7%9A%84-c-%E4%BE%8B%E7%A8%8B/

3、 http://blog.endlesscode.com/2010/06/16/simple-demo-of-streaming-and-pipes/

总结如下：

1、Streaming 是 Hadoop 提供的一个可以使用其他编程语言来进行 MapReduce 来的 API，因为 Hadoop 是基于 Java（由于作者比较擅长 Java，Lucene 和 Nutch 都是出于 Hadoop 的作者）。Hadoop Streaming 并不复杂，其只是使用了 Unix 的标准输入输出作为 Hadoop 和其他编程语言的开发接口，因此在其他的编程语言所写的程序中，只需要将标准输入作为程序的输入，将标准输出作为程序的输出就可以了。在标准的输入输出中，key 和 value 是以 tab 作为分隔符，并且在 reduce 的标准输入中，hadoop 框架保证了输入的数据是经过了按 key 排序的。

2、Hadoop Pipes 是 Hadoop MapReduce 的 C++ 接口。与使用标准输入输出的 Hadoop Streaming 不同（当然 Streaming 也可以用于 C++），Hadoop Pipes 在 tasktacker 和 map/reduce 进行通信时使用的 socket 作为管道，不是标准输入输出，而不是 JNI。Hadoop Pipes 不能运行在 standalone 模式下，所以要先配置成 pseudo-distributed 模式，因为 Hadoop Pipes 依赖于 Hadoop 的分布式缓存技术，而分布式缓存只会在 HDFS 运行的时候才会支持。与 Java 的接口不一样，Hadoop Pipes 的 key 和 value 都是基于 STL 的 string，因此在处理时开发人员需要手动地进行数据类型的转换。

3、从本质上 hadoop pipes 和 hadoop streaming 做的事情几乎一样，除了两者的通信不同，pipes 可以利用 hadoop 的 counter 特性。与 Java native code 比较，Java native code 可以使用实现了 Writable 接口的任何数据类型作为 key/value，而 pipes 和 streaming 就必须通过字符串进行一次转换（通信开销大，存储开销大）。也许正是这个原因，pipes 可能以后会被从 hadoop 中移除。当然，如果计算代价较高，可能 Java native code 并没有 C++ 执行效率高，那么以后可能就写 streaming code 了。pipes 使用的是 byte array，正好可以用 std:string 封装，只是例子里面都是转换成为字符串输入输出。这要求程序员自己设计合理的输入输出方式（数据 key/value 的分段）。

已经得到证实：pipes 已经从 hadoop 中移除。运行 $ ~/hadoop-0.21.0/bin/hadoop，已经看不到 pipe 的这一项了。

使用上的参考：

1、http://developer.yahoo.com/hadoop/tutorial/module4.html#pipes

2、http://code.google.com/p/hypertable/wiki/MapReduceWithHypertable

简单的 Streaming 和 Pipes 示例 （StephenChan's Tech Space）

http://blog.endlesscode.com/ 2010/06/16/simple-demo-of-streaming-and-pipes/

by Stephen posted on 2010/06/16

一、 Hadoop Streaming

Streaming 是 Hadoop 提供的一个可以使用其他编程语言来进行 MapReduce 来的 API，因为 Hadoop 是基于 Java（由于作者比较擅长 Java，Lucene 和 Nutch 都是出于 Hadoop 的作者）。Hadoop Streaming 并不复杂，其只是使用了 Unix 的标准输入输出作为 Hadoop 和其他编程语言的开发接口，因此在其他的编程语言所写的程序中，只需要将标准输入作为程序的输入，将标准输出作为程序的输出就可以了。

在标准的输入输出中，key 和 value 是以 tab 作为分隔符，并且在 reduce 的标准输入中，hadoop 框架保证了输入的数据是经过了按 key 排序的。

下面的示例是用 Python 重写了上一个示例：

```
# max_temperature_map.py
#!/usr/bin/env python
import sys
for linein sys.stdin:
 val= line.strip().split()
 # 分隔年份和温度值，输出到标准输出
 print "%s\t%s"%(val[0], val[1])

# max_temperature_reduce.py
#!/usr/bin/env python
import sys
(last_key, max_val)= (None,0)
for linein sys.stdin:
```

```
                    (key,
                    temp)= line.strip().split('\t')
        if last_keyand last_key != key:
        print "%s\t%s" % (last_key, max_val)
            (last_key, max_val)= (key,int(temp))
        else:
            (last_key, max_val)= (key,max(max_val,int(temp)))

    if last_key:
        print "%s\t%s" % (last_key,
    max_val)
```

“ if last_key and last_key != key 这一行命令主要因为当已经完成了某 (key, value[]) 对的处理后直接输出，然后再重置 (last_key, max_val)进行新的 (key, value[]) 处理。因为 reduce 的输入是经过了排序的，因此 ”if last_key and last_key != key是处理某” key 对应的 value 列表。

执行命令：

```
hadoop jar /home/stephenchan/hadoop-0.20.2 /contrib/streaming/hadoop-0.20.2-streaming.jar
\
-input sample.txt \
-output pyoutput \
        -mapper
        max_temperature_map.py
        \
 -reducer max_temperature_reduce.py
# 不用 MapReduce 其实也相当于下面的命
令：
# cat sample.txt | python max_temperature_map.py | sort | python
max_temperature_reduce.py > pyoutput.txt
```

二、 Hadoop Pipes

Hadoop Pipes 是 Hadoop MapReduce 的 C++接口。与使用标准输入输出的 Hadoop Streaming 不同（当然 Streaming 也可以用于 C++），Hadoop Pipes 在 tasktacker 和 map/reduce 进行通信时使用的 socket 作为管道，不是标准输入输出，而不是 JNI。

Hadoop Pipes 不能运行在 standalone 模式下，所以要先配置成 pseudo-distributed 模式，因为 Hadoop Pipes 依赖于 Hadoop 的分布式缓存技术，而分布式缓存只会在 HDFS运行的时候才会支持。

与 Java的接口不一样，Hadoop Pipes 的 key 和 value 都是基于 STL的 string，因此在处理时开发人员需要手动地进行数据类型的转换。

C++示例代码：

```
/* max_temperature.cpp */
#include <algorithm>
#include <limits>
#include <string>

#include "hadoop/Pipes.hh"
#include "hadoop/TemplateFactory.hh"
#include "hadoop/StringUtils.hh"

class MaxTemperatureMapper :public HadoopPipes::Mapper
{
public:
MaxTemperatureMapper(HadoopPipes::TaskContext& context) {
}

void map(HadoopPipes::MapContext& context) {
std::string line = context.getInputValue();
std::string year = line.substr(0, 4);
```

```
18                  std::string temp = line.substr(5, 3);
19                context.emit(year, temp);
20                }
21              };
22
23              class MaxTemperatureReducer :public HadoopPipes::Reducer {
24              public:
25               MaxTemperatureReducer(HadoopPipes::TaskContext& context) {
26              }
27
28                void reduce(HadoopPipes::ReduceContext&
                context) {
29              int maxValue = 0;
30              while (context.nextValue())
                {
31                maxValue = std::max(maxValue, HadoopUtils::toInt(context.getInputValue()));
32              }
33                context.emit(context.getInputKey(), HadoopUtils::toString(maxValue));
34              }
35              };
36
37              int main(int argc,char *argv[]) {
3                return HadoopPipes::runTask(HadoopPipes::TemplateFactory<MaxTemperatureMapp
8               er, MaxTemperatureReducer>());
39              }
```

makefile 代码：

```
PLATFORM=Linux-i386-32
CC = g++
CPPFLAGS = -m32 -I$(HADOOP_INSTALL)/c++/$(PLATFORM)/include

max_temperature: max_temperature.cpp
$(CC) $(CPPFLAGS) $< -Wall -L$(HADOOP_INSTALL)/c++/$(PLATFORM)/lib -lhadooppipes \
-lhadooputils -lpthread -g -O2 -o $@
```

在使用 make 命令生成了 max_temperature 可执行文件之后，就要使用 ”hadoopfs 将可执行文件和示例数据文件拷贝到伪 dfs 中去。

```
hadoop fs -put max_temperature bin/max_temperature
hadoop fs -put sample.txt temp_sample.txt
hadoop pipes \
-D hadoop.pipes.java.recordreader=true \
-D hadoop.pipes.java.recordwriter=true \
-input temp_sample.txt \
-output cppoutput \
-program bin/max_temperature
# 运行的结果也是保存在伪 dfs 中，使用 "hadoop fs" 来查看
hadoop fs -cat cppoutput/part*
```

Hadoop pipes 编程

火龙果 · 整理
uml.org.cn

Category: Hadoop-MapReduce View: 1,574 阅 Author: Dong
作者 :Dong | 可以转载 , 但必须以超链接形式标明文章原始出处和作者信息及 版权声明
网址 :http://dongxicheng.org/mapreduce/hadoop-pipes-programming/

1. Hadoop pipes 编程介绍

Hadoop pipes 允许 C++ 程序员编写 mapreduce 程序，它允许用户混用 C++ 和 Java 的 RecordReader , Mapper , Partitioner , Rducer 和 RecordWriter 等五个组件。关于 Hadoop pipes 的设计思想，可参见我这篇文章： Hadoop Pipes 设计原理 。

本文介绍了 Hadoop pipes 编程的基本方法，并给出了若干编程示例，最后介绍了 Hadoop pipes 高级编程方法，包括怎样在 MapReduce 中加载词典，怎么传递参数，怎样提高效率等。

2. Hadoop pipes 编程初体验

Hadoop-0.20.2 源代码中自带了三个 pipes 编程示例，它们位于目录 src/examples/pipes/impl 中，分别为 wordcount-simple.cc , wordcount-part.cc 和 wordcount-nopipe.cc 。下面简要介绍一下这三个程序。

（1） wordcount-simple.cc ：Mapper 和 Reducer 组件采用 C++ 语言编写， RecordReader, Partitioner 和 RecordWriter 采用 Java 语言编写，其中， RecordReader 为 LineRecordReader （位于 InputTextInputFormat 中，按行读取数据， 行所在的偏移量为 key , 行中的字符串为 value ）, Partitioner 为 PipesPartitioner , RecordWriter 为 LineRecordWriter （位于 InputTextOutputFormat 中，输出格式为 ” key n ”）

（2） wordcount-part.cc ：Mapper , Partitioner 和 Reducer 组件采用 C++ 语言编写，其他采用 Java 编写

（3）wordcount-nopipe.cc ：RecordReader , Mapper , Rducer 和 RecordWriter 采用 C++ 编写

接下来简单介绍一下 wordcount-simple.cc 的编译和运行方法。

在 Hadoop 的安装目录下，执行下面命令：

```
ant -Dcompile.c++=yes examples
```

则 wordcount-simple.cc 生成的可执行文件 wordcount-simple 被保存到了目录 build/c++-examples/Linux-amd64-64/bin/ 中，然后将该可执行文件上传到 HDFS 的某一个目录下，如/user/XXX/ bin 下：

```
bin/hadoop -put build/c++-examples/Linux-amd64-64/bin/wordcount-simple
/user/XXX/ bin/
```

上传一份数据到 HDFS 的 /user/XXX /pipes_test_data 目录下：

```
bin/hadoop -put data.txt /user/XXX /pipes_test_data
```

直接使用下面命令提交作业：

```
bin/hadoop pipes \

-D hadoop.pipes.java.recordreader=true \

-D hadoop.pipes.java.recordwriter=true \

-D mapred.job.name= wordcount \
```

```
-input /user/XXX /pipes_test_data \
-output /user/XXX /pipes_test_output \
-program /user/XXX/ bin/wordcount-simple
```

3. Hadoop pipes 编程方法

先从最基础的两个组件 Mapper 和 Reducer 说起。

（1） Mapper 编写方法

用户若要实现 Mapper 组件，需继承 HadoopPipes::Mapper 虚基类，它的定义如下：

```
class Mapper: public Closable {

public:

virtual void map(MapContext& context) = 0;

};
```

用户必须实现 map 函数，它的参数是 MapContext ，该类的声明如下：

```
class MapContext: public TaskContext {

public:

virtual const std::string& getInputSplit() = 0;

virtual const std::string& getInputKeyClass() = 0;

virtual const std::string& getInputValueClass() = 0;

};
```

而 TaskContext 类地声明如下：

```
class TaskContext {

public:

class Counter {

……

public:
```

```

    Counter(int counterId) : id(counterId) {}

    Counter(const Counter& counter) : id(counter.id) {}

    ……

    };

    virtual const JobConf* getJobConf() = 0;

    virtual const std::string& getInputKey() = 0;

    virtual const std::string& getInputValue() = 0;

    virtual void emit(const std::string& key, const std::string& value) = 0;

    virtual void progress() = 0;

    …… .

    };
```

用户可以从 context 参数中获取当前的 key，value，progress 和 inputsplit 等数据信息，此外，还可以调用 emit 将结果回传给 Java 代码。

Mapper 的构造函数带有一个 HadoopPipes::TaskContext 参数，用户可以通过它注册一些全局 counter，对于程序调试和跟踪作业进度非常有用：

如果你想注册全局 counter，在构造函数添加一些类似的代码：

```
WordCountMap(HadoopPipes::TaskContext& context) {

inputWords1 = context.getCounter("group", "counter1");

inputWords2 = context.getCounter("group", "counter2");

}
```

当需要增加 counter 值时，可以这样：

```
context.incrementCounter(inputWords1, 1);

context.incrementCounter(inputWords2, 1);
```

其中 getCounter 的两个参数分别为组名和组内计数器名，一个组中可以存在多个 counter。

用户自定义的 counter 会在程序结束时，输出到屏幕上，当然，用户可以用通过 web 界面看到。

（2） Reducer 编写方法

Reducer 组件的编写方法跟 Mapper 组件类似，它需要继承虚基类 public HadoopPipes::Reducer 。

与 Mapper 组件唯一不同的地方时， map 函数的参数类型为 HadoopPipes::ReduceContext ，它包含一个 nextValue() 方法，这允许用于遍历当前 key 对应的 value 列表，依次进行处理。

接下来介绍 RecordReader ， Partitioner 和 RecordWriter 的编写方法：

（3） RecordReader 编写方法

用户自定义的 RecordReader 类需要继承虚基类 HadoopPipes::RecordReader ，它的声明如下：

```
class RecordReader: public Closable {

public:

virtual bool next(std::string& key, std::string& value) = 0;

virtual float getProgress() = 0;

};
```

用户需要实现 next 和 getProgress 两个方法。

用户自定义的 RecordReader 的构造函数可携带类型为 HadoopPipes::MapContext 的参数，通过该参数的 getInputSplit() 的方法，用户可以获取经过序列化的 InpuSplit 对象， Java 端采用不同的 InputFormat 可导致 InputSplit 对象格式不同， 但对于大多数 InpuSplit 对象， 它们可以提供至少三个信息：当前要处理的 InputSplit 所在的文件名，所在文件中的偏移量，它的长度。用户获取这三个信息后，可使用 libhdfs 库读取文件，以实现 next 方法。

下面介绍一下反序列化 InputSplit 对象的方法：

【1】 如果 Java 端采用的 InputFormat 为 WordCountInpuFormat ，可以这样：

```
class XXXReader: public HadoopPipes::RecordReader {

public:

XXXReader (HadoopPipes::MapContext& context) {

std::string filename;

HadoopUtils::StringInStream stream(context.getInputSplit());

HadoopUtils::deserializeString(filename, stream);

……

};
```

【2】 如果 Java 端采用的 InputFormat 为 TextInpuFormat ，可以这样：

```
class XXXReader: public HadoopPipes::RecordReader {

public:

XXXReader (HadoopPipes::MapContext& context) {

std::string filename;

HadoopUtils::StringInStream stream(context.getInputSplit());

readString(filename, stream);

int start = (int)readLong(stream);

int len = (int)readLong(stream);

……

private:

void readString(std::string& t, HadoopUtils::StringInStream& stream)

{

int len = readShort(stream);

if (len > 0) {

// resize the string to the right length

t.resize(len);

// read into the string in 64k chunks

const int bufSize = 65536;

int offset = 0;

char buf[bufSize];

while (len > 0) {

int chunkLength = len > bufSize ? bufSize : len;
```

```

    stream.read(buf, chunkLength);

    t.replace(offset, chunkLength, buf, chunkLength);

    offset += chunkLength;

    len -= chunkLength;

    }

    } else {

    t.clear();

    }

    }

    long readLong(HadoopUtils::StringInStream& stream) {

    long n;

    char b;

    stream.read(&b, 1);

    n = (long)(b & 0xff) << 56 ;

    stream.read(&b, 1);

    n |= (long)(b & 0xff) << 48 ;

    stream.read(&b, 1);

    n |= (long)(b & 0xff) << 40 ;

    stream.read(&b, 1);

    n |= (long)(b & 0xff) << 32 ;

    stream.read(&b, 1);

    n |= (long)(b & 0xff) << 24 ;
```

```

    stream.read(&b, 1);

    n |= (long)(b & 0xff) << 16 ;

    stream.read(&b, 1);

    n |= (long)(b & 0xff) << 8 ;

    stream.read(&b, 1);

    n |= (long)(b & 0xff) ;

    return n;

    }

    };
```

（4） Partitioner 编写方法

用户自定义的 Partitioner 类需要继承虚基类 HadoopPipes:: Partitioner ，它的声明如下：

```
class Partitioner {

public:

virtual int partition(const std::string& key, int numOfReduces) = 0;

virtual ~Partitioner() {}

};
```

用户需要实现 partition 方法和 析构函数。

对于 partition 方法，框架会自动为它传入两个参数，分别为 key 值和 reduce task 的个数 numOfReduces ，用户只需返回一个 0~ numOfReduces-1 的值即可。

（5） RecordWriter 编写方法

用户自定义的 RecordWriter 类需要继承虚基类 HadoopPipes:: RecordWriter ，它的声明如下：

```
class RecordWriter: public Closable {

public:

virtual void emit(const std::string& key,

const std::string& value) = 0;

```

```
};
```

用户自定的 RecordWriter 的构造函数可携带类型为 HadoopPipes::MapContext 的参数，通过该参数的 getJobConf() 可获取一个 HadoopPipes::JobConf 的对象，用户可从该对象中获取该 reduce task 的各种参数，如：该 reduce task 的编号（这对于确定输出文件名有用），reduce task 的输出目录等。

```
class MyWriter: public HadoopPipes::RecordWriter {

public:

MyWriter(HadoopPipes::ReduceContext& context) {

const HadoopPipes::JobConf* job = context.getJobConf();

int part = job->getInt("mapred.task.partition");

std::string outDir = job->get("mapred.work.output.dir");

……

}

}
```

用户需实现 emit 方法，将数据写入某个文件。

4. Hadoop pipes 编程示例

网上有很多人怀疑 Hadoop pipes 自带的程序 wordcount-nopipe.cc 不能运行，各个论坛都有讨论，在此介绍该程序的设计原理和运行方法。

该运行需要具备以下前提：

（1） 采用的 InputFormat 为 WordCountInputFormat，它位于 src/test/ 下的 org.apache.hadoop.mapred.pipes 中

（2） 输入目录和输出目录需位于各个 datanode 的本地磁盘上，格式为：file:///home/xxx/pipes_test (注意，hdfs 中的各种接口同时支持本地路径和 HDFS 路径，如果是 HDFS 上的路径，需要使用 hdfs://host:9000/user/xxx，表示 /user/xxx 为 namenode 为 host 的 hdfs 上的路径，而本地路径，需使用 file:///home/xxx/pipes_test，表示 /home/xxx/pipes_test 为本地路径。例如，bin/hadoop fs -ls file:///home/xxx/pipes_test 表示列出本地磁盘上 /home/xxx/pipes_tes 下的文件)

待确定好各个 datanode 的本地磁盘上有输入数据 /home/xxx/pipes_test/data.txt 后，用户首先上传可执行文件到 HDFS 中：

```
bin/hadoop -put build/c++-examples/Linux-amd64-64/bin/wordcount-nopipe /
```

然后使用下面命令提交该作业：

```
bin/hadoop pipes \

```

```
-D hadoop.pipes.java.recordreader=false \

-D hadoop.pipes.java.recordwriter=false \

-D mapred.job.name=wordcount \

-D
mapred.input.format.class=org.apache.hadoop.mapred.pipes.WordCountInputForm
\

-libjars hadoop-0.20.2-test.jar \

-input file:///home/xxx/pipes_test/data.txt \

-output file:///home/xxx/pipes_output \

-program /user/XXX/bin/wordcount-nopipe
```

5. Hadoop pipes 高级编程

如果用户需要在 mapreduce 作业中加载词典或者传递参数，可这样做：

（1） 提交作业时，用 -files 选项，将词典（需要传递参数可以放到一个配置文件中）上传给各个 datanode ，如：

```
bin/hadoop pipes \

-D hadoop.pipes.java.recordreader=false \

-D hadoop.pipes.java.recordwriter=false \

-D mapred.job.name=wordcount \

-files dic.txt \

… .
```

（2） 在 Mapper 或者 Reducer 的构造函数中， 将字典文件以本地文件的形式打开，并把内容保存到一个 map 或者 set 中，然后再 map() 或者 reduce() 函数中使用即可，如 ：

```
WordCountMap(HadoopPipes::TaskContext& context) {

file = fopen( “ dic.txt ” , "r"); //C 库函数

…… .

}
```

为了提高系能， RecordReader 和 RecordWriter 最好采用 Java 代码实现（或者重用 Hadoop 中自带的），这是因为 Hadoop 自带的 C++ 库 libhdfs 采用 JNI 实现， 底层还是要调用

Java 相关接口，效率很低，此外，如果要处理的文件为二进制文件或者其他非文本文件，libhdfs 可能不好处理。

6. 总结

Hadoop pipes 使 C++ 程序员编写 MapReduce 作业变得可能，它简单好用，提供了用户所需的大部分功能。